# Preservation of Speculative Constant-time by Compilation

SANTIAGO ARRANZ OLMOS, Max Planck Institute for Security and Privacy, Germany
GILLES BARTHE, Max Planck Institute for Security and Privacy, Germany and IMDEA Software Institute, Spain
LIONEL BLATTER, Max Planck Institute for Security and Privacy, Germany
BENJAMIN GRÉGOIRE, Research Centre Inria Sophia Antipolis, France
VINCENT LAPORTE, Centre Inria de l'Université de Lorraine, France

Compilers often weaken or even discard software-based countermeasures commonly used to protect programs against side-channel attacks; worse, they may also introduce vulnerabilities that attackers can exploit. The solution to this problem is to develop compilers that preserve these countermeasures. Prior work establishes that (a mildly modified version of) the CompCert and Jasmin formally verified compilers preserve constant-time, an information flow policy that ensures that programs are protected against cache side-channel attacks. However, nothing is known about preservation of speculative constant-time, a strengthening of the constant-time policy that ensures that programs are protected against Spectre v1 attacks. We first show that preservation of speculative constant-time fails in practice by providing examples of secure programs whose compilation is not speculative constant-time using GCC (GCC -O0 and GCC -O1) and Jasmin. Then, we define a proof-of-concept compiler that distills some of the critical passes of the Jasmin compiler and use the Coq proof assistant to prove that it preserves speculative constant-time. Finally, we patch the Jasmin speculative constant-time type checker and demonstrate that all cryptographic implementations written in Jasmin can be fixed with minimal impact.

## 1 INTRODUCTION

Timing attacks is a class of side-channel attacks that collect and analyze timing behavior of programs to learn about confidential data manipulated during execution [41]. Over the last decades, timing attacks have been used extensively to retrieve secret keys from popular cryptographic implementations, see e.g., [3, 4, 12, 19, 30, 32, 50, 52, 54, 56, 65]. One recipe to protect cryptographic libraries against such attacks is to follow the constant-time (CT) discipline. The CT discipline is simple to state: a program is constant-time if its control flow and memory accesses do not depend on secrets. Unfortunately, writing efficient constant-time code is error-prone. To avoid such errors, developers have access to a broad range of tools for checking that a program is constant-time [13, 31, 37]. Many tools operate at source level. This allows analyzing fine-grained properties (if needed by means of interactive tools) and significantly eases developer feedback. However, this is at the cost of ignoring the possibility of compiler security bugs [29, 64]. They can have disastrous consequences, as demonstrated by the recent side-channel attack of the reference implementation of ML-KEM [51], a post-quantum key encapsulation mechanism under standardization by NIST. Purnal's discovery is an example of a compiler—in this case Clang—undoing programmer's countermeasures in a code snippet with non-trivial control flow—specifically, nested loops. However, the issue also manifests itself in straight line code, i.e., code without conditionals or loops. For instance, Scott [53] recently reports that startling code generated by Fiat-Crypto may be compiled to branching code—in this case, using Clang for a 64-bit RISC-V architecture. An alternative is to check for constant-time at target level, immunizing against compiler bugs compromising security. However, targeting low-level

Authors' addresses: Santiago Arranz Olmos, Max Planck Institute for Security and Privacy, Bochum, Germany, santiago.arranz-olmos@mpi-sp.org; Gilles Barthe, Max Planck Institute for Security and Privacy, Bochum, Germany and IMDEA Software Institute, Madrid, Spain, gilles.barthe@mpi-sp.org; Lionel Blatter, Max Planck Institute for Security and Privacy, Bochum, Germany, lionel.blatter@mpi-sp.org; Benjamin Grégoire, Research Centre Inria Sophia Antipolis, Sophia Antipolis, France, benjamin.gregoire@inria.fr; Vincent Laporte, Centre Inria de l'Université de Lorraine, Nancy, France, vincent.laporte@loria.fr.

programs makes it harder to carry interactive fine-grained analyses and to provide useful feedback to developers. Fortunately, one can obtain the best of the two worlds by proving that the compiler preserves constant-time, i.e., that the compilation of a constant-time program is constant-time. Preservation of constant-time allows us to check for constant-time on source programs, thereby simplifying analysis and programmer feedback, and obtain guarantees on assembly programs, thereby ensuring the absence of compiler induced security vulnerabilities. Recent work shows that preservation of constant-time can be practical. For instance, CompCertCT [15] shows how CompCert [43], a formally verified optimizing C compiler not designed with security in mind, can be made to preserve constant-time with only mild adjustments during instruction selection and assembly generation. Another example is the Jasmin compiler, used to generate efficient cryptographic implementations [5, 6], which has been shown to preserve constant-time [11, 18].

Unfortunately, the constant-time policy is no longer the golden standard for cryptographic libraries. Indeed, there is a long line of work showing that constant-time cryptographic code is vulnerable against transient execution attacks [9, 27, 39, 58–60]. In this paper, we focus on Spectre-PHT [40], a.k.a. Spectre v1, the most basic form of Spectre attack, in which an attacker hijacks the branch predictor in order to force the program to execute a control flow path of their choice, independently of the values of the guards. Spectre-PHT is a powerful attack, allowing, in particular, an attacker to force a program to skip security-critical code. Accounting for Spectre-PHT leads to the notion of speculative constant-time, or SCT, which informally ensures that an attacker with full control over the branch predictor and full ability to observe program leakage cannot learn anything about secrets [26]. Since speculative constant-time is even more error-prone that constant-time, researchers have developed many different tools for checking that a program is SCT [25]. These tools support different protection methods, such as fence insertion [16, 57] and speculative load hardening (SLH) [24, 49, 66]. The most promising approach to date is selective speculative load hardening (selSLH), a refinement of SLH that minimizes the amount of protections inserted in programs based on a fine-grained information flow analysis [9]. Selective SLH is implemented in the Jasmin language [5, 6], in the form of new language primitives that programmers can use at their discretion [10], with the help of an information flow type system that ensures that the program contains sufficiently many and well-placed mitigations. Selective SLH primitives allow programmers to write SCT programs with minimal overhead; for instance, protecting a state-of-the-art ML-KEM implementation against Spectre-PHT incurs a performance penalty of about 1%. Unfortunately, these primitives are added to source programs, creating the risk that the Jasmin compiler produces assembly programs that are not speculative constant-time. The risk of breaking speculative constant-time is not limited to the Jasmin compiler. Indeed, early work by Patrignani and Guarnieri [49] shows that mainstream compilers produce assembly code that is insecure under speculative execution.

*Problem statement and contributions.* The main contribution of this paper is to prove preservation of speculative constant-time for a proof-of-concept compiler inspired by the Jasmin language. We specifically target Spectre-PHT.

As a preliminary step and a motivation for this work, we show that preservation of speculative constant-time fails in practice, either because optimizations tamper with source-level countermeasures, or because they introduce leakage that does not exist at source level. We consider two examples:

- GCC, where we show that manual instrumentation of speculative load hardening in C programs is discarded by GCC -O1. This suggests to use GCC -O0 to preserve security. Unfortunately, we dispel this suggestion by providing an example of a speculative constant-time C program that is compiled using GCC -O0 into an assembly program that is not. In the second case, the

culprit is *spilling*, which introduces new memory accesses and thus opportunities for Spectre attacks;
- Jasmin, where we show an example of a Jasmin program that is speculative constant-time—and, in fact, typable with the type system of [10]—whose compilation is not. In this case, the culprit is memory reuse, and concretely the stack allocation pass that introduces array reuse.

Then, we make the following contributions:

- We develop general results for preservation of speculative constant-time. We lift the well-known notions of forward and backward simulations to the speculative setting, and reestablish in our setting the classical result that forward simulations entail backward simulations assuming progress of the source language and determinism of the target language;
- We define a core language with primitives for protecting against Spectre-PHT, and endow the language with two speculative semantics. The first semantics provides an explicit treatment of initialized values that is used to address the issue with array reuse, and overcomes another incongruity of prior semantics (see Section 5.3). The second semantics imposes additional requirements on directives, which simplifies simulation proofs. We prove the equivalence of these semantics with respect to speculative constant-time;
- We prove preservation of speculative constant-time for a number of compiler passes of interest, including array reuse and array concatenation, which are used in the Jasmin compiler. Then, we prove a composition result and conclude that the combination of all these passes preserves SCT; and
- We adapt the SCT type system of [10] both for our core language and for the Jasmin language. For our core language, we show an end-to-end theorem stating that typable programs are compiled into speculative constant-time programs. For the Jasmin language, we show that the practical impact of our adaptation is very minimal on existing Jasmin implementations. Specifically, we show that all implementations of the libjade library are conveniently patched with little effort and no overhead.

For the sake of concreteness, our presentation discusses exclusively on (selective) speculative load hardening. However, the simplest (but expensive) protection against Spectre-PHT is to insert memory fences. Therefore, it is reasonable to ask if compilers preserve speculative constant-time when programs are protected with fences. All the preservation results of this paper extend *mutatis mutandis* to fences, as the `init_msf()` instruction used in this paper is a form of fence instruction.

*Artifacts.* All our results are formally verified in the Coq proof assistant. The full Coq formalization is available as supplementary material on https://artifacts.formosa-crypto.org/data/sct-preservation.tbz.

## 2 BACKGROUND

Program execution involves intricate interactions between multiple processor components, collectively known as the micro-architecture. These interactions are bidirectional: micro-architectural components are updated during execution, and they influence execution, e.g., through the execution time. For example, the state of caches and predictors is continuously evolving during program execution, e.g., by adding or evicting a cache entry, or by updating branch prediction counters. Additionally, the state of the cache or of branch prediction counters may lead to cache misses and mispredictions, and result in timing differences of executions. Moreover, effects of transient execution may persist after misspeculation has been detected and misspeculated execution has been squashed. While these discrepancies have no influence on the correct execution of programs, attackers can exploit them to recover secrets through side-channel attacks. This section provides a brief review of attacks, policies and countermeasures.

### 2.1 Cache attacks and the constant-time discipline

Cache attacks [19, 50, 56] is a class of side-channel attacks that exploit cache latency, and more concretely the latency between a cache hit and a cache miss, to recover cryptographic keys. Examples of devastating cache attacks include [12, 32, 52, 54, 65].

The constant-time discipline is a programming discipline to protect implementations against cache attacks by requiring that a program's control flow and memory accesses do not depend on secrets. One main appeal of the constant-time discipline is that many cryptographic implementations can be implemented in constant-time with minimal overhead, see [38] for an early example. Another advantage of the constant-time discipline is that it enforces system-level security while simultaneously being expressible neatly in terms of an instrumented semantics of the form $s \xrightarrow{o} s'$ where $o$ is an observation [14]. In the baseline constant-time model, observations are either a step $\bullet$ (for instructions that do not leak), or a boolean value branch $b$ (for branching instructions), or a memory address addr $a\ i$ (for store and load instructions). Then, a program is constant-time w.r.t. some relation $\phi$ on initial states iff any two executions starting from related states yield equal leakage. Constant-time is an instance of observational non-interference, and as such can be checked or enforced using techniques from language-based security. We refer the reader to [13, 31, 37] for a recent overview of some of the main tools for constant-time verification.

### 2.2 Speculative attacks and the speculative constant-time discipline

Unfortunately, constant-time programs remain vulnerable to speculative attacks, and in particular Spectre attacks [40], in which an attacker takes control of speculation mechanisms and makes a victim program leak sensitive information during misspeculated execution.

Speculative attacks can be broadly described by an instrumented semantics of the form $s \xrightarrow[d]{o} s'$ where $s$, $s'$, and $o$ are, as before, two states and an observation, and $d$ is a *directive* that is used for resolving (under-specified) speculative choices and hence models an adversary controlling speculative execution. Given such a speculative semantics, we can define the general notion of *speculative constant-time* (SCT for short), with respect to a relation $\phi$ on initial states: it states that a program does not leak sensitive information in spite of speculative execution.

Several approaches have been proposed to mitigate speculative attacks, and in particular Spectre-PHT, which exploits branch prediction. The simplest one is to insert a fence after every branch, effectively blocking any misspeculated execution. Another approach is index masking, which prevents speculative out-of-bounds accesses by ensuring that every array access is of the form $a[e \bmod n]$, where $n$ is the size of the array. However, both approaches have a significant performance overhead. A more efficient approach, named *speculative load hardening* (or SLH for short), creates control flow dependencies (which may be abused speculatively) through data dependencies (which are not affected by speculation) [24, 49, 66]. In this way, leaking misspeculated execution steps are blocked until the end of the speculation window. SLH induces quite a large run-time overhead. A more refined approach to SLH, called *selective* SLH (selSLH for short) exploits knowledge about which data is secret to reduce the need for protection: in many cases, it is unnecessary to detect misspeculation.

## 3 MOTIVATING EXAMPLES

This section illustrates issues with preservation of speculative constant-time for GCC, a widely used compiler that is not designed with security in mind, and Jasmin, a domain-specific compiler designed with security in mind.

```
1  #define update_msf(msf, e) (e ? msf : -1)
2  #define protect(x, msf) (x | msf)
3
4  uint64_t init_msf() {
5      asm volatile ("lfence" : : : "memory");
6      return 0;
7  }
8
9  uint64_t load(uint64_t* p, uint64_t i) {
10     uint64_t msf = init_msf();
11     uint64_t x = 0;
12     if (i < 10) {
13         msf = update_msf(msf, i < 10);
14         x = p[i];
15         x = protect(x, msf);
16     } else {
17         msf = update_msf(msf, !(i < 10));
18     }
19     return p[x];
20 }
```

```
1  load:
2      lfence
3      cmpq $9, %rsi
4      ja .L
5      movq (%rdi,%rsi,8), %rax
6      movq (%rdi,%rax,8), %rax
7      retq
8  .L:
9      xorl %eax, %eax
10     movq (%rdi,%rax,8), %rax
11     retq
```

Fig. 1. A function with SLH mitigations, before and after compilation with GCC 12.3.0 at optimization level 1. The array p has size ten.

### 3.1 Case study I: GCC

Spectre-PHT attacks typically exploit unsafe memory accesses during speculative execution. The essence of speculative load hardening is to mask such memory accesses such that they do not leak secrets. To this end, it is necessary to instrument programs so that they can track whether execution is misspeculating. This is done using a register *msf* (the *misspeculation flag* or MSF) to track misspeculation, and updating this register after each branching statement to check if execution proceeded into the correct branch or misspeculated. Figure 1 shows an example of a program instrumented with speculative load hardening, taken from Figure 3a in [10]. The first instruction of the procedure load sets the *msf* register to zero using the init_msf instruction, one of the three selSLH instructions from [10]. This instruction inserts a fence, which has the effect of stopping speculative execution, and sets the misspeculation flag to zero, to indicate that the program is executing sequentially—i.e., it has not misspeculated. Then, the flag is updated immediately after entering each branch of a conditional. This is carried out using the instruction update_msf. Critically, this instruction updates the MSF using a conditional move, which, contrary to conditional jumps, is not affected by speculation. This way, update_msf ensures that the register *msf* accurately tracks misspeculation. Based on this, we can protect vulnerable memory accesses using the protect instruction, which is the last selSLH primitive from [10]. This instruction masks values if execution has misspeculated, so that arrays are accessed to a default index in that case. Overall, the instrumentation ensures that the program from Figure 1 is speculative constant-time.

Speculative load hardening mitigations are redundant computations in sequential execution; the GCC compiler (version 12.3.0) detects this at optimization level 1. It notices that the updates to the msf variable are redundant and therefore removes them, as witnessed by the produced assembly listing displayed on the right of Fig. 1: the compiler removes the protects and the initialization of the MSF. As a result, the output assembly is not speculative constant-time.

It is often tempting to address security issues introduced by compilers simply by turning off optimizations. However, Fig. 2 provides an example of a speculative constant-time program that

```
1  uint8_t* tbl;
2
3  uint8_t example(uint64_t sec, uint64_t pub) {
4      uint64_t r = 0;
5      uint64_t p[1] = { pub };
6      if (pub != pub) { p[1] = sec; }
7      return tbl[r];
8  }
```

```
example:
        ...
        movq   %rdi, -8(%rbp)     ;; push sec
        movq   $0, -24(%rbp)      ;; init r
        ...
        je     .L
        movq   -8(%rbp), %rax     ;; read sec
        movq   %rax, -24(%rbp)    ;; write r
.L:     movq   tbl(%rip), %rax    ;; load tbl
        movq   -24(%rbp), %rcx    ;; load r
        movb   (%rax,%rcx), %al   ;; tbl[r]
        ...
```

Fig. 2. Spilling does not preserve speculative constant-time.

is compiled by GCC -O0 into an assembly program that is not. Indeed, the listing on the left of Fig. 2 presents a sequentially safe program (note the out-of-bounds access inside the never-taken then-branch) that is speculative constant-time (even under misspeculation, the value of r is zero). When compiled with GCC -O0, we get the listing on the right of Fig. 2. By default, the compiler spills function arguments on the stack, allowing an attacker to exploit unsafe memory accesses to overwrite the local variable r with secret value sec, even though this was not possible in the source program.

### 3.2 Case Study II: Jasmin

Jasmin [5, 6] is a framework for high-speed, high-assurance, cryptographic software. Its main components are the Jasmin language and the Jasmin compiler. The language is based on the "assembly in the head" paradigm; it features a combination of low-level constructs, such as vectorized instructions, and high-level constructs, such as structured control flow (conditionals and loops), and zero-cost abstractions, such as explicit variable names and functional arrays. These high-level constructs and zero-cost abstractions vastly simplify reasoning about program correctness and program security. At the same time, the Jasmin framework is designed to generate efficient assembly code, so the Jasmin compiler eliminates source-level abstractions. In the case of arrays, the compiler checks that arrays are used linearly, which allows modifying them in-place and ensures that there are no run-time copies. The compiler also minimizes stack usage through array reuse, i.e., allowing two arrays with different live ranges to use the same region of the stack.

Recently, in [10], the Jasmin language was extended with a small set of language constructs to support selSLH, and a type system for verifying that programmers make correct use of them. The type system allows using different countermeasures, i.e., it is able to type check programs protected with fence insertion, index masking, speculative load hardening, and selective speculative load hardening. Moreover, this approach has proven effective for protecting optimized cryptographic primitives with a negligible performance overhead. Unfortunately, there are examples of Jasmin programs that are typable with the type system, and hence speculative constant-time with respect to the definition of [10], but are compiled to assembly programs that are not.

We start by illustrating the problem with a constructed example. Consider the programs from the left of Fig. 3. The first program (top left), the source program, uses two arrays. The first array s is used to store a secret value. Then, we have a conditional that is never taken, with an uninitialized load from the second array p to a register r. Finally, the value of the register is leaked through a leaky instruction, which we model abstractly as leak r. The program is safe and speculative constant-time.

```
r = 0;
s[0] = sec;
if (false) { r = p[0]; }
leak r;
```

```
r = 0;
s[0] = sec;
if (false) { r = s[0]; }
leak r;
```

```
export fn sign(reg u64 s m len k) {
    _ = #init_msf();
    reg u128 key;
    key = preprocess_secret_key(k);
    reg u64 i; stack u8[N] state;
    i = 0; while (i < N) { state[i] = 0; i += 1; }
    i = 0; while (i + B < len) {
        state = absorb(state, m, i);
        i += B;
}
```

Fig. 3. Array reuse does not preserve speculative constant-time.

In this situation, the array reuse optimization of Jasmin detects that the arrays have different live ranges, so we can use a single array, leading to the insecure program at the bottom. Indeed, the load instruction now accesses a secret value, which is leaked by the next instruction. Note that the program is sequentially safe *only* because the load instruction is unreachable, since it speculatively accesses an uninitialized memory cell. Unfortunately, in presence of array reuse, uninitialized memory cells may hold secret values, causing the leak instruction to expose the secret. The reader may wonder why the branch is not removed by dead code elimination; however, the example is written for the sake of readability, and it is easy to modify the source program so that the compiler does not detect that the guard is always false. The solution to this problem is to modify the semantics of source programs so that every load from an uninitialized memory cell is adversarially controlled, and thus an attacker can load a value from a memory cell of their choice. Under such a semantics, the source program is no longer speculative constant-time and requires the protections that allow preservation of SCT. We emphasize that our solution does not require modifying the semantics of assembly programs, for which we ultimately seek guarantees. This is because the semantics of assembly programs assumes a single memory that is initialized at the beginning of the program.

As previously indicated, our example is artificial. However, it reflects a real-world issue, as illustrated by the program on the right of Fig. 3, which presents the high-level structure of a signing procedure. This function receives as argument a pointer `m` to a *public* message together with its length `len` and a pointer `k` to a secret key. First, the key is loaded from memory in the `preprocess_secret_key` function; then the message is hashed, one block at a time, incrementally updating the `state` variable. Said state is kept on the local stack and initialized to a public initialization vector (here only zeros). As the state only depends on public data (the message, its length, and the initialization vector), it should be typed as public. In fact, the type system for SCT for Jasmin [10] would infer that it is public even during misspeculated executions. However, if the `preprocess_secret_key` function stores parts of the secret key into local variables that get allocated by the compiler at the same stack position as part of the state, and if a misspeculated execution bypasses the initialization of the state, suddenly the state may depend, at assembly level, on secret data. This highlights that in order to take into account the effects of the compilation process (here allocating local variables into stack frames and sharing the same address space for secret and public data) the notion of security must accommodate uninitialized memory and conservatively assume that uninitialized memory holds secret data.

$$\textsc{nil}\ \frac{}{m \xrightarrow[\epsilon]{\epsilon}{}^{*}\ m} \qquad \textsc{cons}\ \frac{m \xrightarrow[d]{o} m' \qquad m' \xrightarrow[ds]{os}{}^{*}\ m''}{\mathsf{Running}\ m \xrightarrow[d \cdot ds]{o \cdot os}{}^{*}\ m''}$$

Fig. 4. Multi-step speculative execution semantics.

## 4 SECURITY

In this section, we recast the definition of SCT in an abstract setting, and extend the well-known notions of forward and backward simulations to establish preservation of SCT.

A compiler often handles programs written in several programming languages: source, target, and intermediate representations. The formalism below keeps programming languages fairly unspecified and enables relating programs written in several languages. Syntactically speaking, a programming language is modeled as a set $\mathcal{L}$ of programs. We assume given a set $\mathcal{I}$ of inputs, common to all programming languages. Inputs describe how a program interacts with (reads from) its environment. For concreteness, we will simply consider that inputs are lists of values, to be taken as arguments by the program entry point.

So as to be able to define the security notions of interest, we consider semantics that are *instrumented* with observations and directives. Observations model data that leaks through side-channels and may be available to an adversary. Directives model the ability of an adversary to influence the program execution and in particular decisions that are taken speculatively; in essence, directives resolve all non-deterministic choices that may occur during program execution.

The semantic of a programming language is given by a set $\mathcal{M}$ of execution states (sometimes referred to as *machines*), a set $\mathcal{D}$ of directives, a set $\mathcal{O}$ of observations, an initialization function $\mathcal{L} \times \mathcal{I} \to \mathcal{M}$, and a small-step relation written $s \xrightarrow[d]{o} r$ where $s$ is a state, $d$ a directive, $r$ a machine *configuration*, and $o$ an observation. The configuration that is reached as a result of an execution step is either a final outcome (Final $f$) or an other execution state from which the execution might continue (Running $s'$).

The small-step relation is extended to a many-steps relation between machine configurations reading a list of directives and producing a list of observations (see Fig. 4). When there are no directives, execution makes no step and produces no observation (rule NIL). Otherwise, a first step is taken according to the first directive, then execution continues, accumulating the observations in a list (rule CONS).

The SCT security notion is defined relative to a relation on input values, often noted $\phi$, analogous to the *low-equivalence* relation usually found in the definition of constant-time (or non-interference in general). It expresses what part of the input is public. Often, we consider simple relations such as "the first argument is public", which may be formally defined as $i\ \phi_0\ i' := \mathsf{hd}(i) = \mathsf{hd}(i')$. This formalism is much more general and can express many precise conditions. Consider for instance a program that checks that user input (its first argument) matches a predefined secret code (its second argument). The security policy that this program complies with is expressed by the relation $(x_1, y_1)\ \phi_1\ (x_2, y_2) := x_1 = x_2 \wedge (x_1 = y_1 \iff x_2 = y_2)$, which states that the value $x$ of the first input is public and that the tested condition is also public (i.e., declassified).

*Definition 1 ($\phi$-SCT).* Let $\phi \in \mathcal{P}(\mathcal{I} \times \mathcal{I})$ be a relation on inputs. We note $\phi$-SCT the set of programs $P$ (in any language) satisfying

$$\forall i_1\ i_2\ D\ O_1\ O_2\ c_1\ c_2,\ i_1\ \phi\ i_2 \wedge \mathsf{Running}\ P(i_1) \xrightarrow[D]{O_1}{}^{*}\ c_1 \wedge \mathsf{Running}\ P(i_2) \xrightarrow[D]{O_2}{}^{*}\ c_2 \implies O_1 = O_2.$$

Note that we conveniently conflate a program with its initialization function.

A machine configuration $c$ is said to be *sequentially safe* when any configuration reachable from $c$ can make a step unless it is final or misspeculating. Therefore, the programming language semantics is assumed to provide primitives to detect when a configuration is final and when it is misspeculated.

As we will generally only consider executions starting from inputs related by some $\phi$, we will assume that this low-equivalence relation also enforces any sequential safety preconditions.

We now turn to preservation. A transformation $\mathcal{T}$ that relates programs (maybe written in different languages) is said to preserve SCT when $\forall P\ Q\ \phi, P\ \mathcal{T}\ Q \wedge P \in \phi\text{-SCT} \implies Q \in \phi\text{-SCT}$.

So as to establish this property, we notice that given a program transformation, the observation trace of the target program can usually be explained (i.e., computed) from an observation trace of the source program. Since the semantics cover speculative behaviors and are driven by directives, in order to pick the source execution that explains the target execution, source directives have to be constructed from target directives.

Theorem 1 (Secure compilation; forward case). *Let $P$ and $Q$ be programs. If there exist two functions $T_d$ and $T_o$ such that*

$$\forall D\ i\ O\ s, P(i) \xrightarrow[T_d(D)]{O}{}^{*}\ s \implies \exists t, Q(i) \xrightarrow[D]{T_o(O)}{}^{*}\ t$$

*then for any $\phi$, if $P$ is $\phi$-SCT then $Q$ is $\phi$-SCT.*

A weaker version of this result starts from the target execution, but explicitly requires sequential safety of the source program.

Theorem 2 (Secure compilation; backward case). *Let $P$ and $Q$ be programs. If there exist two functions $T_d$ and $T_o$ such that*

$$\forall D\ i\ O_t\ t, P(i)\ \textit{is sequentially safe} \wedge Q(i) \xrightarrow[D]{O_t}{}^{*}\ t \implies \exists O_s\ s, P(i) \xrightarrow[T_d(D)]{O_s}{}^{*}\ s \wedge O_t = T_o(O_s)$$

*then for any $\phi$, if $P$ is $\phi$-SCT then $Q$ is $\phi$-SCT.*

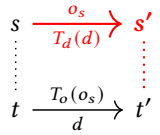
Note that both versions require a form of sequential safety of the source program, and that the first one implies preservation of sequential safety (i.e., for every input $i$ such that $P(i)$ is sequentially safe, then $Q(i)$ is also sequentially safe). In order to compose transformations that preserve security, it is therefore required to justify sequential safety after every step, which may be done either as part of the security preservation proof or as part of a usual correctness proof.

These theorems remain complex to apply. As is often the case in certified compilation, verifying a program transformation entails identifying a relation between execution states of the source and target programs that is maintained during their executions. Such a relation is called a simulation relation and proving it amounts to establishing a simulation diagram. One of the simplest forms of such a diagram arises when source and target executions progress at the same pace and when directives and observations are transformed in a similar way at every step.

*Definition 2 (BW lock-step simulation diagram).* Given two programs $P$ and $Q$, a relation $\sim$ between source and target machines is a *lock-step simulation* with directive transformer $T_d$ and observation transformer $T_o$ when:

(1) initial states are in relation, i.e., $\forall i, P(i) \sim Q(i)$; and
(2) it is preserved at every step (in the picture, hypotheses are in black and conclusions in **bold red**), i.e.,

$$\forall s\ t\ d\ o_t\ t', s \sim t \wedge t \xrightarrow[d]{o_t} t' \wedge s \text{ is sequentially safe} \implies \exists s'\ o_s, s \xrightarrow[T_d(d)]{o_s} s' \wedge o_t = T_o(o_s) \wedge s' \sim t'.$$




In many situations, this diagram is too strong. A simple relaxation of this diagram allows the transformers to depend on the history of the executions, as realized by the list of (target) directives that have been followed from an initial state to reach the target state $t$ under consideration.

To cover a wide scope of program transformations, we will use in the rest of this paper a few more general diagrams. They all enjoy the fundamental soundness property of ensuring secure compilation (and therefore SCT-preservation).

Theorem 3 (Soundness of BW lock-step simulation). *Given $P$ and $Q$ two programs and $\sim$, $T_d$, and $T_o$ satisfying the BW lock-step simulation diagram, the hypothesis of Theorem 2 holds, with $T_d$ and $T_o$ naturally extended to lists of directives and list of observations (respectively).*

Proof. By induction on the sequence of target directives. □

## 5 LANGUAGE

We instantiate the framework of the previous section with a core language with SLH primitives.

### 5.1 Syntax

Programs are written in a core imperative language with arrays and SLH constructs. The syntax of instructions and code is as follows:

$$\begin{aligned} i &::= x = e \mid x = a[e] \mid a[e] = x \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \\ &\mid \texttt{init\_msf}() \mid \texttt{update\_msf}(e) \mid x = \texttt{protect}(x) \\ c &::= \epsilon \mid i; c \end{aligned}$$

where $e$ are expressions, $x$ register variables, and $a$ array variables. We assume that each array comes with its size $|a|$. We also assume a distinguished register *msf*.

### 5.2 Semantics

*Sequential semantics.* The sequential semantics of programs is mainly standard. However, our semantics accounts for uninitialized values, and requires that arrays are initialized before their use.

Machines are triples $\langle c, \rho, \mu, ms \rangle$ consisting of the code being executed, a register map mapping register names to optional values, and a memory mapping pairs of arrays and valid indices to optional values. We use optional values to track initialization: $\mu(a, j) = \bot$ means that the array $a$ is not initialized at index $j$ (and $j$ is within bounds, i.e., $0 \le j < |a|$). The operational semantics of programs is given by the relation $m \to m'$; the rules are standard and omitted. Note that the semantics implicitly assumes that programs are sequentially safe, i.e., all sequential array accesses are within bounds, and arrays are initialized before being read.

*Instrumented semantics.* Our instrumented semantics lets execution produce observations. Observations correspond to the kind of side-channels we consider, we will use the classic constant-time leakage model:

$$\text{Obs} ::= \bullet \mid \text{branch } b \mid \text{addr } a\ i.$$

The observation $\bullet$ corresponds to the observation of a single execution step but no leakage, branch $b$ indicates that the condition of a conditional evaluated to $b$, addr $a$ $i$ that a memory access to array

$a$ in position $i$ occurred. The instrumented semantics of programs is given by the relation $m \xrightarrow{o} m'$; the rules are standard and omitted.

*Speculative semantics.* Directives model the attacker's power to influence speculation and are given by the syntax

$$\mathsf{Dir} ::= \mathsf{step} \mid \mathsf{force}\ b \mid \mathsf{mem}\ a\ i.$$

The directive step is used for the execution of atomic instructions, i.e., assignments and SLH instructions. The directive force $b$ is used for the execution of conditional and loops and forces execution of the $b$ branch, reflecting the attacker's control of the branch predictor. Finally, the directive mem $a$ $i$ is used for loads and stores and forces these instructions to read from or write to the adversarially chosen address $(a, i)$ if the instruction performs an unsafe or uninitialized memory access, reflecting our conservative choice to give attacker's full control over memory accesses in these cases. If the access is in bounds and the cell is initialized, the directive is ignored. We assume that $i$ is always in bounds of $a$.

The speculative semantics of programs $m \xrightarrow[d]{o} m'$ and its multi-step version are defined in Figure 5 and Figure 4 respectively. We briefly comment on the rules.

The ASSIGN rule models the usual semantics of an assignment, where the only possible directive is step and a the observation is always $\bullet$.

The N-LOAD rule models the semantics of loads. The index must be in bounds of the array, and the corresponding cell initialized. The retrieved value is stored into a register, and the address (not the value) is leaked via the observation addr $a$ $i$. As mentioned before, the directive is ignored in this case. The dual of this rule is S-LOAD, which models the semantics of loads under misspeculation. This is the case when the index is out-of-bounds or the cell is uninitialized. The directive mem $b$ $j$ causes the instruction to load a value from memory address $b[j]$ instead, and store it in a register. Execution still leaks observation addr $a$ $i$. Note that the rule requires that the misspeculation status is $\top$.

The N-STORE rule models the semantics of sequential stores. Similarly to the N-LOAD case, the index must be in bounds and the cell initialized. On the other hand, the S-STORE rule is analogous to the S-LOAD rule.

The following three rules are for the selSLH instructions. The INIT-MSF is a speculation fence, which means that the machine must be in sequential execution, and assigns the initial value NOMASK to a distinguished register $msf$. The UPDATE-MSF is a conditional update of $msf$: it updates its value to MASK if and only if the condition is false. The PROTECT rule shows how a value is masked with $msf$. If the value of $msf$ is MASK, the output is the default value MASK, and otherwise just the value of the input variable.

The rules for control flow instructions, COND and WHILE, always follow the adversary's choice of branch given in the force $b$ directive. The observation branch $b'$ leaks the evaluation of the condition, and the misspeculation status is updated accordingly. Finally, the NIL and CONS rules define the semantics for code as the reflexive transitive closure of the one for instructions, accumulating the directives and observations.

LEMMA 1 (DETERMINISM). *The semantics is deterministic. That is, given two single-step executions* $m \xrightarrow[d]{o_1} m_1$ *and* $m \xrightarrow[d]{o_2} m_2$, *we have* $o_1 = o_2$ *and* $m_1 = m_2$.

### 5.3 Comparison with prior semantics

Our semantics differs from [10] in two ways. First, in the S-LOAD rule. The clause for uninitialized memory accommodates undesirable interactions between uninitialized memory and memory reuse.

ASSIGN

$$\frac{}{\langle x = e; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{step}]{\bullet} \langle c,\ \rho[x \leftarrow [\![ e ]\!]_\rho],\ \mu,\ ms\rangle}$$

N-LOAD

$$\frac{[\![ e ]\!]_\rho = i \qquad i \in [0, |a|) \qquad \mu(a, i) = v}{\langle x = a[e]; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\textbf{mem } \boldsymbol{b\ j}]{\text{addr } a\ i} \langle c,\ \rho[x \leftarrow v],\ \mu,\ ms\rangle}$$

S-LOAD

$$\frac{[\![ e ]\!]_\rho = i \qquad i \notin [0, |a|) \boldsymbol{\vee \mu(a, i) = \bot} \qquad j \in [0, |b|) \qquad \mu(b, j) = v}{\langle x = a[e]; c,\ \rho,\ \mu,\ \top\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ i} \langle c,\ \rho[x \leftarrow v],\ \mu,\ \top\rangle}$$

N-STORE

$$\frac{[\![ e ]\!]_\rho = i \qquad i \in [0, |a|)}{\langle a[e] = x; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\textbf{mem } \boldsymbol{b\ j}]{\text{addr } a\ i} \langle c,\ \rho,\ \mu[(a, i) \leftarrow \rho(x)],\ ms\rangle}$$

S-STORE

$$\frac{[\![ e ]\!]_\rho = i \qquad i \notin [0, |a|) \qquad j \in [0, |b|)}{\langle a[e] = x; c,\ \rho,\ \mu,\ \top\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ i} \langle c,\ \rho,\ \mu[(b, j) \leftarrow \rho(x)],\ \top\rangle}$$

INIT-MSF

$$\frac{}{\langle \texttt{init\_msf}(); c,\ \rho,\ \mu,\ \bot\rangle \xrightarrow[\text{step}]{\bullet} \langle c,\ \rho[msf \leftarrow \text{NOMASK}],\ \mu,\ \bot\rangle}$$

UPDATE-MSF

$$\frac{v = \text{if } [\![ e ]\!]_\rho \text{ then } \rho(msf) \text{ else MASK}}{\langle \texttt{update\_msf}(e); c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{step}]{\bullet} \langle c,\ \rho[msf \leftarrow v],\ \mu,\ ms\rangle}$$

PROTECT

$$\frac{v = \text{if } \rho(msf) = \text{MASK then MASK else } \rho(y)}{\langle x = \texttt{protect}(y); c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{step}]{\bullet} \langle c,\ \rho[x \leftarrow v],\ \mu,\ ms\rangle}$$

COND

$$\frac{[\![ e ]\!]_\rho = b'}{\langle \texttt{if } e \texttt{ then } c_\top \texttt{ else } c_\bot; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{force } b]{\text{branch } b'} \langle c_b; c,\ \rho,\ \mu,\ ms \vee (b \neq b')\rangle}$$

WHILE

$$\frac{[\![ e ]\!]_\rho = b' \qquad c' = \text{if } b \text{ then } c_0; \texttt{while } e \text{ do } c_0; c \text{ else } c}{\langle \texttt{while } e \text{ do } c_0; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{force } b]{\text{branch } b'} \langle c',\ \rho,\ \mu,\ ms \vee (b \neq b')\rangle}$$

Fig. 5. Single-step speculative execution semantics. Differences with earlier semantics are highlighted in **bold red**.

As a consequence, the program from Figure 3 is speculative constant-time according to [10], but is not SCT according to our definition.

Second, in the N-LOAD and N-STORE rules. Our semantics uses a directive mem $b$ $j$ for all load and store operations. This is in contrast with [10] where load and store operations may also use

ERR-N-LOAD

$$\frac{[\![ e ]\!]_\rho = i \qquad i \in [0, |a|) \qquad \mu(a,i) = v \qquad m' = \begin{cases} \langle c,\ \rho[x \leftarrow v],\ \mu,\ ms\rangle & \text{if } a,i = b,j \\ \mathrm{err_M} & \text{otherwise} \end{cases}}{\langle x = a[e]; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ i} m'}$$

ERR-N-STORE

$$\frac{[\![ e ]\!]_\rho = i \qquad i \in [0, |a|) \qquad m' = \begin{cases} \langle c,\ \rho,\ \mu[(a,i) \leftarrow \rho(x)],\ ms\rangle & \text{if } a,i = b,j \\ \mathrm{err_M} & \text{otherwise} \end{cases}}{\langle a[e] = x; c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ i} m'}$$

Fig. 6. Semantics with error.

a step directive. However, using different directives depending on whether the execution step is sequential or speculative leads to an unhappy interaction with the notion of SCT defined above. As an example, the program `if (false) then x = a[sec % 2]`, where $|a|$ is 1 and `sec` is a secret, is speculative constant-time under the semantics from [10]. This is because all executions of this program *under the same directives* in states that coincide everywhere except on `sec` produce the same observations: if the directive for the load instruction is step, then if `sec` is even we get the observation addr $a$ 0, while if it is odd, we get not observation—the execution is stuck—and thus there is nothing to verify. The converse happens if the directive is mem $b$ $j$, since if `sec` is even the access will be in bounds and only allow the step directive:

$$\textit{sec} \text{ is even} \quad \langle x = a[sec\%2], \ldots\rangle \xrightarrow[\text{step}]{\text{addr } a\ 0} \ldots \qquad \langle x = a[sec\%2], \ldots\rangle \xrightarrow[\text{mem } b\ j]{} \not\longrightarrow$$

$$\textit{sec} \text{ is odd} \quad \langle x = a[sec\%2], \ldots\rangle \xrightarrow[\text{step}]{} \not\longrightarrow \qquad \langle x = a[sec\%2], \ldots\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ 0} \ldots$$

Since no directive causes the machines to generate different observations, the program is SCT. However, this contradicts our intuition that the program leaks and its not SCT. The semantics presented in the current work does not suffer from this, since memory operations always take a mem $b$ $j$ directive, execution leaks the parity of *sec* and the program is thus not SCT.

### 5.4 Error semantics

Our proofs of preservation use both the semantics described above and a refinement that is more informative in the rules for memory operations, but equivalent for the purposes of showing SCT. This semantics replaces the N-LOAD and N-STORE rules, as shown in Fig. 6, to enforce that if the memory operation is sequential, in bounds, and initialized, the directive is not ignored but required be exactly mem $a$ $i$. It introduces a distinguished "error machine" $\mathrm{err_M}$, to indicate this mismatch. The advantage of this semantics is that both the control flow and the memory accesses of an execution are uniquely determined by the sequence of directives. The disadvantage is that it is not immediately clear that restricting the semantics in this way faithfully models speculative execution. However, since the definition of speculative constant-time quantifies over all sequences of directives, we are able to prove that for the purposes of proving SCT they are equivalent. That is, a program is speculative constant-time under the semantics presented in Fig. 5 if and only if it is speculative constant-time under the error semantics (i.e., with the rules from Fig. 6).

Lemma 2 (Equivalence of error semantics). *A program is $\phi$-SCT if and only if it is $\phi$-SCT$_{\mathrm{err_M}}$.*

## 6 PRESERVATION OF SCT BY COMPILATION

In this section, we consider nine compiler passes: constant folding, constant propagation, dead assignment elimination, loop peeling, dead branch elimination, array reuse, register allocation, array concatenation, and linearization, and prove that they preserve SCT. In broad terms, the proofs of all passes of our compiler involve two main components: first, we ensure a degree of functional correctness in sequential execution such that the evaluation of expressions leaked by the program is the same. Second, we prove that the compiler does not introduce new opportunities for speculative leakage, by defining a showing that target leakage can be computed by source leakage. Our proofs require sequential safety of the source in some passes, as discussed. Informally, sequential safety of a machine $m$ means that all machines reachable by executing $m$ are either final, misspeculating, or can perform a step. Formally,

$$\mathsf{seqsafe}(m) \coloneqq \forall \vec{d}\ \vec{o}\ m', m \xrightarrow[\vec{d}]{\vec{o}}{}^{*} m' \implies m'_c = \epsilon \vee m'_{ms} \vee \exists d\ o\ m'', m' \xrightarrow[d]{o} m''.$$

All the proofs presented in this section use backward simulations and more particularly Theorem 3, instantiated with the semantics with error machines from Section 5.4. By convention, we lift every simulation relation $\sim$ to optional machines in the conclusion of each backward lemma as follows:

$$s \sim_{\mathrm{err_M}} t \coloneqq \begin{cases} \top & \text{if } t \text{ is } \mathrm{err_M}, \\ s \neq \mathrm{err_M} \wedge s \sim t & \text{otherwise.} \end{cases}$$

In other words, an error in the target means that there is nothing to prove, otherwise we must show that the source is not an error and that they are related.

We now establish preservation of speculative constant-time for individual passes of the compiler. For each pass, we first briefly explain its goals and mechanics and then turn to the proof of preservation of SCT; in particular, we provide details about the simulation and the directive and observation transformers for one-step execution. Note that many of our compiler passes are defined abstractly, and more concretely relative to oracles that perform standard analyses, such as liveness analysis for dead code elimination or valid renamings for register allocation. Such an abstract view of compiler passes is compatible and even beneficial with our goal to explore the feasibility of preserving SCT. Moreover, instantiating these oracles with concrete analyzers has no implication on security.

Finally, note that all but the last optimizations are source-to-source translations of the language described in Section 5. On the other hand, linearization produces a program in a language that features the same basic instructions but changes the representation of a program to a directed graph, as explained below.

*Constant folding.* This pass replaces composite constant expressions in the program with their evaluation. For instance, the expression `x + (3 - 1)` goes to `x - 2`. It does not introduce or remove instructions, it replaces only expressions in the program. As this pass is quite simple, we do not need to parameterize over any analyses.

The simulation relation for this pass states that the code of the target machine is the compilation of the source's and that their states are equal. The transformers for directives and observations are the identity, so we only need to show that the machines are in lock-step and that their leakage is the same. The former comes from the pass not adding or removing instructions, the latter from leakage containing only concrete values—the leakage of `x = a[3 - 2]` and `x = a[1]` is the same, addr $a$ 1.

*Constant propagation.* This pass replaces variables with their values when they are constant. For example, `x = 3; y += x + 4` gets transformed into `x = 3; y += 3 + 4`. It does not introduce or remove

instructions; it replaces the expressions in the program—specifically, in assignments, memory operations, MSF updates, conditionals, and loops. This pass is parameterized by an oracle that attaches to each program point a map from variables to optional values, meaning that at this point, a variable is guaranteed to hold the associated value—e.g., in the example from before, in the second instruction, we would get a map $\{x \longrightarrow 3\}$.

We prove the simulation under the assumption that the oracle is functionally correct, the standard requirement for a correct compiler. The simulation relation states that the code of the target is the compilation of the source's, that their states are the same, and that the map given by the oracle is correct w.r.t. the state—i.e., that variables have their predicted values. The directive and observation transformers are the identity, so we prove that they are in lock-step and the evaluation of leaked expressions coincide.

*Dead assignment elimination.* This pass removes some instructions that are not needed for the program. Specifically, it removes assignments, MSF updates, and protects when they do not contribute to the result or leakage of the program; it does not remove any other instruction. Remark that we can remove MSF updates (resp. protects) without compromising security only if the MSF is unused (resp. protected value is not leaked). It is parameterized by an oracle that returns the set of needed variables at each program point; we guarantee the correctness of this information with a verified checker. The checker is as expected, where, for instance, let $S$ be the set of needed variables after an assignment $x = e$, the set of needed variables before the assignment is $(S \setminus \{x\}) \cup \mathsf{FV}(e)$ in the case that $x \in S$, and $S$ otherwise.

This simulation relation states that the code of the target is the compilation of the source's, the register maps coincide on the set of needed variables, and the memories and misspeculation statuses are the same, that is

$$t_c = (\!|\, s_c,\, S \,|\!), \qquad t_\rho =_S s_\rho, \qquad t_\mu = s_\mu, \qquad \text{and} \qquad t_{ms} = s_{ms}$$

where $S$ is given by the oracle and $\rho =_S \rho'$ means that $\rho$ and $\rho'$ coincide on the registers in $S$. Its transformers remove some step directives and $\bullet$ observations in the target, corresponding to dead instructions

$$(\!|\, d_t, c \,|\!)_{\mathsf{Dir}} \coloneqq \begin{cases} \mathsf{step} \cdot (\!|\, d_t, c' \,|\!)_{\mathsf{Dir}} & \text{if } c = i; c' \wedge \mathsf{dead}(i, c'), \\ d_t & \text{otherwise,} \end{cases} \qquad (\!|\, \vec{o_s} \,|\!)_{\mathsf{Obs}} \coloneqq \mathsf{last}(\vec{o_s})$$

where $\mathsf{dead}(i, c)$ means that the written variables of $i$ are not needed by $c$, as determined by the oracle. This means that the source will perform several steps for each target step, for the dead assignments preceding the target instruction: if the target performs a step under directive $d_t$ and observation $o_t$, the source needs $\mathsf{step} \cdot \ldots \cdot \mathsf{step} \cdot d_t$ and $\bullet \cdot \ldots \cdot \bullet \cdot o_t$.

*Loop peeling.* This pass performs one unrolling of some loops. Specifically, it transforms loops `while` $e$ `do` $c$ into conditionals `if` $e$ `then` $c'$; `while` $e$ `do` $c'$ `else` $\epsilon$ where $c'$ is the transformation of $c$. We assume that certain loops are annotated as candidates for unrolling and perform this peeling only on them—this annotation may come, for instance, from an analysis that expects the loop range to be known at compile time. We can unroll some loops by combining this pass with constant propagation, folding, and dead code elimination. Since the semantics is preserved exactly, we need no constraints on the oracle.

The simulation relation states that the target code is either the compilation of the source's or a peeled loop after its first iteration

$$\frac{}{\epsilon \sim \epsilon} \qquad \frac{c_s \sim c_t}{i_s; c_s \sim (\!|\, i_s,\, O \,|\!); c_t} \qquad \frac{c'_s \sim c'_t \qquad c_s \sim c_t}{\texttt{while}\ e\ \texttt{do}\ c'_s; c_s \sim \texttt{while}\ e\ \texttt{do}\ c'_t; c_t}$$

where $O$ is the oracle indicating which loops should be peeled. Note that the requirement for code is not simply $t_c = (\!| s_c, O |\!)$ as usual; this is because, in the first iteration, the source executes a loop and the target executes a conditional (the compilation of said loop), however, for all the following iterations both execute the same loop. The simulation also requires that their states are equal. Since the directive for both conditionals and loops is force $b$, and we do not change any instruction that updates the state, the directive and observation transformers are the identity, and the machines are in lock-step. As evinced by the simulation for code, there are three subcases for loops: a loop that was not peeled, a peeled loop in its first iteration, and one after the first iteration.

*Dead branch elimination.* This pass simplifies conditionals and loops whose guards are constant. That is, the pass transforms conditionals $\texttt{if}\ b\ \texttt{then}\ c_\top\ \texttt{else}\ c_\bot$ into the transformation of $c_b$ and loops $\texttt{while}\ \bot\ \texttt{do}\ c$ into $\epsilon$. It does not use oracles, as it simply matches expressions on exact constants.

The simulation relation is straightforward: the code of the target is the compilation of the source's, and the states are equal. The directive and observation transformers must account for the removed branches, akin to dead assignment elimination. That is, when the target executes the then-branch of $\texttt{if}\ \top\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots$, the source needs to do an extra step to enter said branch, namely force $\top$. Consequently, the transformers depend on the code of the source machine to be able to compute the prefix force $b$ directives. Hence, we define $\mathsf{pre}_c(\cdot)$ to compute this prefix: it returns a pair of a boolean, indicating whether all instructions have been removed until this point, and the prefix of removed force $b$ directives until now

$$\mathsf{pre}_i(i) \coloneqq \begin{cases} (r, \mathsf{force}\ b \cdot \vec{d}) & \text{if } i = \texttt{if}\ b\ \texttt{then}\ c_\top\ \texttt{else}\ c_\bot \wedge \mathsf{pre}_c(c_b) = (r, \vec{d}), \\ (\bot, \mathsf{force}\ \bot) & \text{if } i = \texttt{while}\ \bot\ \texttt{do}\ c, \\ (\bot, \epsilon) & \text{otherwise,} \end{cases}$$

$$\mathsf{pre}_c(c) \coloneqq \begin{cases} (r, \vec{d}_i \cdot \vec{d}_c) & \text{if } c = i; c' \wedge \mathsf{pre}_i(i) = (\top, \vec{d}_i) \wedge \mathsf{pre}_c(c') = (r, \vec{d}_c,) \\ (\bot, \vec{d}_i) & \text{if } c = i; c' \wedge \mathsf{pre}_i(i) = (\bot, \vec{d}_i), \\ (\top, \epsilon) & \text{otherwise,} \end{cases}$$

$$(\!| d_t, c |\!)_{\mathsf{Dir}} \coloneqq \vec{d}_c \cdot d_t \quad \text{where } (\_, \vec{d}_c) = \mathsf{pre}_c(c).$$

Intuitively, the directive transformer consumes as many conditionals and loops with constant guards from the code of the source machine as possible, for which it issues force $b$ directives, and ends the sequence with the target directive. Schematically, if the target executes under a directive $d_t$ and produces an observation $o_t$, the source will do so under directives $\mathsf{force}\ b_1 \cdot \ldots \cdot \mathsf{force}\ b_n \cdot d_t$ and observations $\mathsf{branch}\ b_1 \cdot \ldots \cdot \mathsf{branch}\ b_n \cdot o_t$. The transformers for observations and many-step executions are identical to the ones for dead assignment elimination.

We split the proof backward SCT simulation of this pass into two lemmas. Let $s$ and $t$ be a source and target machines, such that they are in the simulation. We will first show that if the pass does not remove the first instruction of $s$, they both step under the same directive and produce the same observation. Then, we show that it is always possible for $s$ to execute a (possibly empty) sequence of force $b$ steps until we fall in the first case. Note that the last lemma, when the source stutters, need not say anything about the source observations. Indeed, these are irrelevant since the last execution step of the source always comes from the execution in the first lemma, when both the source and target step, and $(\!| \vec{o_s} |\!)_{\mathsf{Obs}} = \mathsf{last}(\vec{o_s}) = o_t$.

*Array reuse.* This pass aims to reduce (stack) memory usage by reusing dead arrays. As an illustration, let $a$ and $b$ be two arrays of equal size in a program such that its first part does not use $b$ and its second part does not use $a$—that is, the two arrays are never simultaneously alive. In this situation,

| Source | Target | uninit | $\mathsf{rtbl}^{-1}(c)$ | $\mathsf{Dir}_t$ | $\mathsf{Dir}_s$ | $\mathsf{wtbl}(c)$ |
|---|---|---|---|---|---|---|
| $a[0] = s;$ | $c[0] = s;$ | $\{a, b\}$ | – | mem $c$ 0 | mem $a$ 0 | $[a, a]$ |
| $b[1] = 1;$ | $c[1] = 1;$ | $\{b\}$ | $a$ | mem $c$ 1 | mem $b$ 0 | $[\boldsymbol{a}, a]$ |
| if $false$ then | if $false$ then | $\emptyset$ | $b$ | force $\top$ | force $\top$ | $[a, \boldsymbol{b}]$ |
| $x = b[0]$ | $x = c[0]$ | $\emptyset$ | $b$ | mem $c$ 0 | mem $a$ 0 | $[a, b]$ |

Fig. 7. An example of array reuse where it is challenging to define the directive transformer, where the size of arrays $a$, $b$, and $c$ is two.

the compiler can safely replace all uses of $a$ and $b$ with a new array of the same size, $c$, thus requiring one memory region instead of two. The first two columns of Fig. 7, "Source" and "Target", illustrate this example. Array reuse, therefore, consists of simply renaming array variables in loads and stores at annotated points. We restrict this pass to replace only arrays of equal size for simplicity.

This pass requires an oracle, a *reuse (or renaming) table*, denoted rtbl, which indicates the renaming that the compiler should perform. With this notation, array reuse consists of replacing each array $a$ in the program by $\mathsf{rtbl}(a)$. Remark that the point of this pass is for rtbl *not* to be injective. Naturally, we need to ensure that this table is valid w.r.t. the source program, which means that if the compiler wants to rename $a$ and $b$ into $c$, as in the example, no cell of $b$ can be initialized while its counterpart in $a$ is live. Consequently, we require an analysis that computes two pieces of information at each program point: a set, denoted uninit, of the uninitialized arrays at this point; and a mapping, denoted $\mathsf{rtbl}^{-1}$, from target arrays to source arrays, such that each target coincides with its mapping on its initialized cells. If the instruction at the current program point is $a[e] = x$, then the correctness of the analysis entails that either $a \in \mathsf{uninit}$ or $\mathsf{rtbl}^{-1}(\mathsf{rtbl}(a)) = a$. Moreover, the analysis after this store (i.e., for the next instruction) removes $a$ from uninit and asserts $\mathsf{rtbl}^{-1}(\mathsf{rtbl}(a)) = a$. A similar condition holds for loads, except that the analysis after the instruction is unchanged.

The conditions presented above are sufficient to prove correctness but not preservation of SCT. Figure 7 illustrates the problem; recall that the first two columns depict the example program from before, where source arrays $a$ and $b$ get renamed into $c$. Note that this program is sequentially safe, and the renaming is sequentially correct, but the source performs an uninitialized memory access under misspeculation. The second and third columns show the static information from the compile-time analysis needed to check the validity of the transformation. The "uninit" column shows the set of uninitialized source arrays at each program point, and the "$\mathsf{rtbl}^{-1}(c)$" column shows the source array that was renamed into $c$ at that point, if any. The fourth and fifth columns show dynamic information corresponding to an example execution. The "$\mathsf{Dir}_t$" column lists directives for the target program—note that the branch misspeculates. At the end of this target execution, $x$ will hold the value $s$. In order to explain this behavior at the source level, we need to construct the source directives shown in the "$\mathsf{Dir}_s$" column. For safe loads, we can use $\mathsf{rtbl}^{-1}$ to transform the first two directives. However, the static information we have is not enough to transform the last directive: we need to remember that the first cell of $c$, which at that point is the renaming of $b$, has the same value as the first cell of $a$. This information is not available from the static data alone, as there may be more than two arrays renamed to $c$, and, in fact, may depend on the specific execution.

Nevertheless, there is still hope for our proof: using the semantics with error presented in Section 5.4, we can determine which source array corresponds to unsafe memory accesses from the previous directives of the execution. In our example, we know that at the time we executed the previous write with target directive mem $c$ 0, the analysis gave $\mathsf{rtbl}(a) = c$, so the source array

that coincides with $c$ at position 0 is $a$. This is what the *write table* does, denoted "wtbl($c$)", which we show in the last column: we keep track, for each cell of each target array, which source array contains the corresponding value (from the last write to the cell). Contrary to $\mathsf{rtbl}^{-1}$, this information cannot be computed statically but can be derived—in the semantics with error—from the history of directives. In Fig. 7, we see that after the first store, with mem $c$ 0 and $\mathsf{rtbl}(a) = c$, we update $\mathsf{wtbl}(c)$ with $a$ in the first cell (in **bold red**). After the second store, with mem $c$ 1 and $\mathsf{rtbl}(b) = c$, we update it with $b$ in the second cell. Other instructions do not update this table.

In this manner, we can parameterize the directive transformer with a write table, and compute the next one after each execution step, from the target directive, the source code, the reuse table, and the current write table, all of which are public data. The precise definition of the transformer is

$$\langle\!|\, d, c, \mathsf{wtbl} \,|\!\rangle_{\mathsf{Dir}} = \begin{cases} \mathsf{mem}\ \mathsf{wtbl}(b)[j]\ j & \text{if } d = \mathsf{mem}\ b\ j \wedge c = x = a[e]; c', \\ \mathsf{mem}\ \mathsf{rtbl}^{-1}(b)\ j & \text{if } d = \mathsf{mem}\ b\ j \wedge c = a[e] = x; c' \wedge \mathsf{rtbl}(a) \neq b, \\ \mathsf{mem}\ a\ j & \text{if } d = \mathsf{mem}\ b\ j \wedge c = a[e] = x; c' \wedge \mathsf{rtbl}(a) = b, \\ d & \text{otherwise.} \end{cases}$$

The transformer for observations is straightforward: it renames a source observation addr $a$ $i$ into the target observation addr $\mathsf{rtbl}(a)$ $i$ and leaves the other observations unchanged.

The simulation relation of array reuse is that the code of the target is the compilation of the source's, that the states are equal modulo the renaming of arrays given by the oracle, and that the information from the oracle is correct. Concretely, we say that a source machine $s$ is in relation with a target machine $t$ with respect to wtbl if

$$\mathsf{valid}(s_c), \qquad t_c = \langle\!|\, s_c, \mathsf{rtbl} \,|\!\rangle, \qquad t_\rho = s_\rho, \qquad s_\mu, t_\mu \models \mathsf{uninit}, \mathsf{rtbl}^{-1}, \mathsf{wtbl}, \qquad \text{and} \qquad t_{ms} = s_{ms}$$

where $s_\mu, t_\mu \models \mathsf{uninit}, \mathsf{rtbl}^{-1}, \mathsf{wtbl}$ means that the memories validate the information from the analyses and the write table.

*Register allocation.* This pass renames the register variables of a program. It does not introduce or remove instructions. It can be used, for instance, to make a program require only architectural registers (such as RAX, RDI, RSI, etc.), but also in standard compiler passes such as inlining, which we do not consider in the present work. It has two components: an untrusted function that performs the renaming and a verified checker. The function takes and returns a program, and the checker ensures they are alpha equivalent. We do not implement the renaming, only the checker—i.e., the pass is parameterized over the renaming function.

The simulation relation states that the code of the target machine is a valid renaming of that of the source and that the states are equal modulo this renaming (note that the renaming is of register variables only, so the condition on states for memories and misspeculation statuses is equality). Specifically, we require $t_\rho(x) = s_\rho(O(x))$ for each register variable $x$, where $O$ is the renaming function at the current point. The directive and leakage transformers are the identity. Thus, we show that the machines are in lock-step and that source and target expressions evaluate to the same value.

*Array concatenation.* This pass renames array accesses to use one large array instead of many small ones. That is, instead of using $a$ and $b$ with $|a| = 2$ and $|b| = 3$, we can use $c$ with $|c| = 5$ and replace all instances of $a[e]$ with $c[e]$ and $b[e]$ with $c[e + 2]$. Compilers usually feature a pass like this to introduce stack frames with the local data of a function. This pass is parameterized by an oracle that maps source arrays to target arrays and offsets: in the example from before, $a$ goes to $(c, 0)$ and $b$ to $(c, 2)$.

As usual, we must ensure some level of functional correctness. Under sequential execution, we need two conditions on the oracle for this pass to be correct: that the target array can fit the source array, that is, that for every source array $a$, target array $c$, and offset $n$, we ensure that

$$O(a) = (c, n) \implies [n, n + |a|) \subseteq [0, |c|),$$

and that source arrays do not overlap inside the target, that is, that for every two source arrays $a$ and $b$, target array $c$, and offsets $n$ and $m$, we require that

$$O(a) = (c, n) \land O(b) = (c, m) \implies [n, n + |a|) \cap [m, m + |b|) = \emptyset.$$

Unfortunately, these correctness conditions are insufficient to prove preservation of speculative constant-time. Since an adversary might be able to speculatively read from the target array $c$ at any offset, we must also require that this pass is no more than a renaming of memory regions and that we are not exposing "new" memory for the attacker to read. That is to say, we need every cell of each target array to correspond to a cell in a source array (put differently, there must be no holes between source arrays): for each target array $c$ and position $i$ we need that

$$i \in [0, |c|) \implies \exists a\ n, O(a) = (c, n) \land i \in [n, n + |a|).$$

This restriction looks problematic, for instance, if the compiler needs to align source arrays, but it is not so. As explained before, arrays in our model are disjoint memory regions; this condition states that the compiler must introduce no new regions. It is always possible to consider that the source program has more regions than it uses, uninitialized "dummy" arrays, that will serve as padding between other source arrays.

Using these three hypotheses, we can show that there exists an inverse function $O^{-1}$ of the oracle, satisfying that for every source array $a$, target array $c$, offset $n$, and position $j$,

$$j \in [n, n + |a|) \land O(a) = (c, n) \implies O^{-1}(c, j) = (a, j - n).$$

We can now define the simulation relation for this pass, which says that the code of the target is the compilation of the source's and that their states are equal modulo the renaming of memory addresses from the oracle. More precisely, we ask that their register maps and misspeculation statuses are equal, and that $t_\mu(c, j) = s_\mu(O^{-1}(c, j))$ for each target array $c$ and position $j$. The directive and observation transformers rename mem $a$ $i$ directives and addr $a$ $i$ observations according to the oracle, and leave the other directives and observations unmodified

$$(\!|\, d_t \,|\!)_{\mathsf{Dir}} := \begin{cases} \mathsf{mem}\ a\ i & \text{if } d_t = \mathsf{mem}\ c\ j \\ & \text{and } O^{-1}(c, j) = (a, i), \\ d_t & \text{otherwise,} \end{cases} \quad (\!|\, o_s \,|\!)_{\mathsf{Obs}} := \begin{cases} \mathsf{addr}\ c\ (n + i) & \text{if } o_s = \mathsf{addr}\ a\ i \\ & \text{and } O(a) = (c, n), \\ o & \text{otherwise.} \end{cases}$$

*Linearization.* This pass removes structured control flow (i.e., conditionals and loops) by using a graph representation of the program. The target language has the same basic instructions as the source, e.g., assignments, loads, and stores, but the edges of the graph capture control flow. A program in the target language is a directed graph where the nodes are instructions and out-neighbors are successor instructions; instructions have exactly one successor except conditional jumps, which have two. Programs also have two distinguished labels, entry and exit, to start and halt execution. Machines in this language are similar to source machines, with a program label instead of code, i.e., $\langle \ell,\ \rho,\ \mu,\ ms \rangle$. We denote $\langle \ell,\ \rho,\ \mu,\ ms \rangle_{pc}$ for the program counter of this machine, $\ell$.

We characterize this pass as a relation between source and target programs, where $\ell : i_t \to \vec{\ell}$ means that label $\ell$ points to the target instruction $i_t$, and its successors are $\vec{\ell}$, and $\ell : (\!|\, i_s \,|\!) \to \vec{\ell}$ similarly but for the linearization of the source instruction $i_s$. As the specification of this pass is a

relation, our compiler is parametric on the exact way linearization is done as long as it satisfies the relation. The nontrivial cases of the compilation scheme are

$$\frac{\begin{array}{rl}\ell : \texttt{if } e \texttt{ jump} & \longrightarrow \ell_\top, \ell_\bot \\ \ell_\top : \quad (\!| c_\top |\!) & \longrightarrow \ell' \\ \ell_\bot : \quad (\!| c_\bot |\!) & \longrightarrow \ell'\end{array}}{\ell : (\!| \texttt{if } e \texttt{ then } c_\top \texttt{ else } c_\bot |\!) \longrightarrow \ell'} \qquad \frac{\begin{array}{rl}\ell : \texttt{if } e \texttt{ jump} & \longrightarrow \ell_{body}, \ell' \\ \ell_{body} : \quad (\!| c |\!) & \longrightarrow \ell\end{array}}{\ell : (\!| \texttt{while } e \texttt{ do } c |\!) \longrightarrow \ell'}.$$

This pass should be followed by one that places basic blocks sequentially instead of having a graph representation. Such a pass simply needs to find program points with more than one in-neighbor and introduce an unconditional jump in one of the branches. It would preserve speculative constant-time since unconditional jumps do not allow speculation or leak information.

The simulation relation for this pass requires that the label of the target machine points to either the compilation of the code of the source machine or the exit label of the program

$$c \sim pc := \begin{cases} pc = \ell_{\mathsf{exit}} & \text{if } c \text{ is } \epsilon, \\ pc : (\!| i |\!) \longrightarrow \ell \wedge c' \sim \ell & \text{if } s_c \text{ is } i; c', \end{cases}$$

and that their states are equal. The directive and observation transformers are the identity, hence we show that the machines are in lock-step.

*Composition.* Composing the proof of preservation of each pass yields a proof of preservation of SCT for the whole compiler, assuming that the source program is sequentially safe. In our formalization, we assume that passes preserve sequential safety: this is a common property usually satisfied by correct compilers that can be proved using standard methods. The proof of preservation of SCT imposes no restrictions on the order in which our passes are composed.

Proposition 1 (Compiler preservation of SCT). *Let $\phi$ be a relation on inputs and $p_s$ a source program, and $p_t$ its compilation. If $p_s$ is $\phi$-SCT and sequentially safe, $p_t$ is $\phi$-SCT.*

## 7 TYPE SYSTEM

This section defines an information flow type system that enforces SCT with respect to our new semantics. The type system is taken from [10], but the notion of typable program puts an additional requirement on the initial type.

For simplicity of exposition, we present the type system in a simplified form. In particular, our presentation does not allow for label polymorphism and does not follow the constraint-based style used in type inference.

*Type system.* We now describe an information flow type system that ensures that observations do not depend on secrets, for any adversarial choice of directives—i.e., that the program is SCT. Typing judgments are of the form $\Sigma, \Gamma \vdash i : \Sigma', \Gamma'$, where $\Gamma$ and $\Gamma'$ are security environments that map registers and arrays to pairs of security levels taken from a 2-elements lattice $\{\mathsf{Public}, \mathsf{Secret}\}$ with the usual order $\mathsf{Public} \leq \mathsf{Secret}$. The components of these pairs respectively track the confidentiality level under sequential execution and under all possible speculative executions. By convention, we write $\Gamma_n$ and $\Gamma_s$ for the first and second component of $\Gamma$. Given that speculative execution encompasses sequential execution, we implicitly assume the invariant $\Gamma_n \leq \Gamma_s$. Moreover, $\Sigma$ and $\Sigma'$ are misspeculation flag (MSF) types taken from the grammar

$$\Sigma ::= \mathsf{unknown} \mid \mathsf{updated} \mid \mathsf{outdated}(e).$$

Informally, MSF types capture partial information about the way the MSF flag tracks misspeculation. We briefly explain each type in turn. The MSF type unknown means that nothing is known

$$\frac{\Gamma \vdash e : \tau \qquad x \notin \mathsf{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = e : \Sigma, \Gamma[x \leftarrow \tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{Public} \qquad x \notin \mathsf{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = a[e] : \Sigma, \Gamma[x \leftarrow \langle \Gamma(a)_n, \mathsf{Secret} \rangle]}$$

$$\frac{\begin{array}{c}\Gamma \leq \Gamma' \qquad \Gamma \vdash e : \mathsf{Public} \\ \Gamma(x) \leq \Gamma'(a) \qquad \forall b \neq a, \Gamma(x)_s \leq \Gamma'(b)_s\end{array}}{\Sigma, \Gamma \vdash a[e] = x : \Sigma, \Gamma'}$$

$$\frac{\Gamma = \{\, v : \langle \Gamma(v)_n, \Gamma(v)_n \rangle \mid \text{for each } v \,\}}{\Sigma, \Gamma \vdash \mathtt{init\_msf}() : \mathsf{updated}, \Gamma'}$$

$$\frac{}{\mathsf{outdated}(e), \Gamma \vdash \mathtt{update\_msf}(e) : \mathsf{updated}, \Gamma}$$

$$\frac{\tau = \langle \Gamma(x)_n, \Gamma(x)_n \rangle}{\mathsf{updated}, \Gamma \vdash y = \mathtt{protect}(x) : \mathsf{updated}, \Gamma[y \leftarrow \tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{Public} \qquad \Sigma|_e, \Gamma \vdash c_\top : \Sigma', \Gamma' \qquad \Sigma|_{!e}, \Gamma \vdash c_\bot : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ c_\top\ \mathtt{else}\ c_\bot : \Sigma', \Gamma'}$$

$$\frac{\Gamma \vdash e : \mathsf{Public} \qquad \Sigma|_e, \Gamma \vdash c : \Sigma, \Gamma}{\Sigma, \Gamma \vdash \mathtt{while}\ e\ \mathtt{do}\ c : \Sigma|_{!e}, \Gamma}$$

$$\frac{}{\Sigma, \Gamma \vdash \epsilon : \Sigma, \Gamma}$$

$$\frac{\Sigma, \Gamma \vdash i : \Sigma_i, \Gamma_i \qquad \Sigma_i, \Gamma_i \vdash c : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash i; c : \Sigma', \Gamma'}$$

$$\frac{\begin{array}{c}\Sigma_0, \Gamma_0 \vdash c : \Sigma_0', \Gamma_0' \qquad \Sigma_0 \subseteq \Sigma \\ \Sigma' \subseteq \Sigma_0' \qquad \Gamma \leq \Gamma_0 \qquad \Gamma_0' \leq \Gamma'\end{array}}{\Sigma, \Gamma \vdash c : \Sigma', \Gamma'}$$

$$\mathsf{FV}(\Sigma) := \begin{cases} \mathsf{FV}(e) & \text{if } \Sigma \text{ is outdated}(e) \\ \emptyset & \text{otherwise} \end{cases} \qquad \Sigma|_e := \begin{cases} \mathsf{outdated}(e) & \text{if } \Sigma \text{ is updated} \\ \mathsf{unknown} & \text{otherwise} \end{cases}$$

$$\Sigma \subseteq \Sigma' := \Sigma = \mathsf{unknown} \vee \Sigma = \Sigma' \qquad \Gamma \leq \Gamma' := \forall x, \Gamma(x) \leq \Gamma'(x)$$

Fig. 8. Typing rules and auxiliary definitions.

about misspeculation. The MSF type updated means that the register *msf* correctly captures misspeculation status, i.e., *msf* is equal to MASK if execution is misspeculating and NOMASK otherwise. The type outdated($e$) means that the register *msf* will correctly capture misspeculation status after being updated with the expression $e$.

Figure 8 presents the core typing rules. We refer to [10] for further explanations. In addition to these rules, the type system features specific rules for safe loads and stores. These rules relax the typing constraints of loads and stores, under the assumption that they are speculatively safe. Informally, a load or store is speculatively safe if for every execution, including speculative executions of the program, the index of the load or store remains within bounds. Under this assumption, one does not need to set the speculative type of the target register of a load to Secret. Similarly, one does not need to update the speculative type of all arrays when typing a speculatively safe store. It is possible to adapt these rules to our setting, by strengthening the notion of speculative safety to encompass correct initialization.

*Soundness.* For every program $p$ and relation $\phi$ be an equivalence relation that defines secrets and public inputs, we let $\Gamma_{p,\phi}$ be the security environment that maps to $\langle \mathsf{Secret}, \mathsf{Secret} \rangle$ every register and array that does not store a public input. A key observation is that, whenever $\Gamma_{p,\phi}$ is used as an initial type, the typing rules ensure that only variables and arrays initialized with public data can be public. This suffices to prove that the type system ensures SCT.

Proposition 2 (Soundness). *If* $\mathsf{unknown}, \Gamma_{p,\phi} \vdash c : \Sigma', \Gamma'$, *for some* $\Sigma'$ *and* $\Gamma'$, *then* $c$ *is* $\phi$*-speculative constant-time.*

*End-to-end security.* By composing Proposition 2 and Proposition 1, we obtain the following end-to-end theorem.

Theorem 4 (End-to-end security). *Let $p_s$ be a source program, and $p_t$ its compilation. If $p_s$ is sequentially safe and* unknown, $\Gamma_{p_s} \vdash p_s : \Sigma', \Gamma'$, *for some $\Sigma'$ and $\Gamma'$, then $p_t$ is $\phi$-speculative constant-time.*

## 8 APPLICATIONS TO JASMIN AND LIBJADE

In this section, we explore the significance of our results for the Jasmin language. We discuss preservation of speculative constant-time for its compiler, as well as the impact of our findings on high-assurance cryptographic software written in Jasmin, including the libjade library.

### 8.1 Preservation of speculative constant-time

Table 1 lists the principal passes of the Jasmin compiler and briefly discusses preservation of speculative constant-time; whenever possible, we contrast the Jasmin pass with one of the compiler passes considered in this paper. The overall conclusion is that most passes are either similar to the ones considered in this work or clearly preserve speculative constant-time.

Unfortunately, it remains a significant engineering endeavor to prove preservation of speculative constant-time for the Jasmin compiler. The first reason for this is that the proof of preservation of (sequential) constant-time [18] is a formalization (of over ten thousand lines according to *loc. cit.*) that uses complex objects, including structured leakage transformers. Moreover, this formalization has not been updated with respect to the latest versions of the Jasmin compiler. We conjecture that a proof of preservation of SCT would be even larger and more complex. Second, the compiler and the proof of compiler correctness are under major refactoring, in particular, to switch from the current big-step semantics to an alternative semantics based on interaction trees [63].

### 8.2 Impact on Jasmin implementations

The Jasmin framework has been used to write efficient and formally verified implementations of many cryptographic primitives, including the hash function SHA3 and the key encapsulation mechanism ML-KEM, formerly Kyber. Most of these implementations are typable with the type system of [46]. However, we have seen that this type system accepts Jasmin programs that compile into assembly that is not SCT. To avoid the problem, we have patched the type system to make additional requirements on the initial type, see Section 7. To understand the impact of this requirement, we have adapted the implementation of the Jasmin type system so that it enforces our additional constraints. Pleasingly, all implementations except ML-KEM require no modifications. The type system rejects ML-KEM because the rejection sampling procedure uses local arrays, which must be initially declared as secret in our type system. We manually inspected the sources of failure and concluded that they correspond to reading public values in initialized arrays. The type system accepts the program after being instructed that these reads are legitimate, using the `declassify` annotation of the Jasmin language. As declassification has no computational content, the modifications have no effect on performance.

We need to manually instruct the type system that these loads are legitimate because, as discussed in Section 7, the type of uninitialized variables must be secret. The type checker implements some heuristics to improve on this, specifically for registers and stack variables, since once it finds a write to these, it can assume that they are initialized. For arrays, it is more complex; see, for instance the example of the listing on the right of Fig. 3. In the case of ML-KEM, manual code inspection reveals that the array in question must be initialized before rejection sampling.

Table 1. Jasmin compiler passes and their effect on leakage (we refer to the passes presented in this work when applicable). We expect that all passes preserve SCT.

| Jasmin pass | Transformers for directives and observations |
|---|---|
| Inlining | Add step directives and • observations before and after function calls. |
| Function pruning | The identity (this pass removes unreachable parts of the program). |
| Constant propagation | Remove some force $b$ directives and branch $b$ observations (this pass is a combination of constant folding, propagation, and dead branch elimination from this work). |
| Dead code elimination | Remove some step directives and • observations (the same as dead assignment elimination from this work). |
| Unrolling | The identity (this pass is repeated loop peeling). |
| Register array expansion | Source mem $a$ $i$ directives and addr $a$ $i$ observations become step and • in the target when $a$ is a register array. |
| Instruction selection | Add step directives and • observations for complex assignments. |
| SLH instruction selection | The identity. |
| Inline propagation | The identity (this is constant propagation on inline variables only). |
| Stack allocation | Renaming of mem $a$ $i$ directives and addr $a$ $i$ observations (this is a combination of array reuse and (partial) array concatenation from this work). |
| Register allocation | The identity (the same as register allocation from this work). |
| Linearization | Add some step directives and • observations (a combination of the linearization pass of this work and a pass that inserts intermediate direct unconditional jumps). |
| Stack zeroization | Add mem RSP $i$ directives and addr RSP $i$ observations, where $i$ is constant. |
| Tunneling | Remove some step directives and • observations. |
| Assembly generation | Add step directives and • observations for complex operations. |

## 9 OTHER SPECTRE VARIANTS

So far, we have focused on the Spectre-PHT, a.k.a. Spectre v1, variant of Spectre. We now review other variants of Spectre and discuss preservation of speculative constant-time in this context. We refer the reader to recent surveys [22, 25] for further background and pointers to the literature.

*Spectre v2.* Spectre v2 is a variant of Spectre that exploits indirect branch speculation to force execution to jump to arbitrary program points. Retpoline [36] (and its variants [8, 35]) is a software-based countermeasure that replaces indirect jumps by return instructions, the protection comes from delaying speculative execution until the speculation window closes. It could be of general interest to study preservation in presence of retpolines. However, this is out-of-scope of our work, as our primary target is the Jasmin language, which does not have indirect jumps.

*Spectre v4.* Spectre v4 is a variant of Spectre that exploits speculative store bypass to force memory reads to retrieve stale values. The sole software-only countermeasure against Spectre v4 is fence insertion; in particular, it is not possible to generalize speculative load hardening to Spectre v4, because there is no good mechanism for programs to track misspeculation. Alternatively, one can use CPU flag to disable speculative store bypass. We believe that the results of this paper extend to speculative store bypass. However, this requires extending the semantics of programs with a

memory buffer, e.g., in the style of [16]. This is out-of-scope of our work, as our primary target is the Jasmin language, which currently targets CPUs with the SSBD flag on.

*Spectre-RSB.* Spectre RSB [42, 62] is a variant of Spectre that exploits the return stack buffer to force function calls to return to arbitrary program points. There exist different countermeasures against Spectre-RSB. The most relevant countermeasure is based on a combination of SLH instructions and program rewriting to rewrite calls and returns by jump tables made of cascading sequences of conditionals [46]. The soundness of their countermeasure relies on a preservation argument between an idealized semantics and the real semantics of programs. Hence their approach could in principle be integrated in our framework. However, their preservation result requires source programs to be well-typed, which would force to make program rewriting as the first pass of our compiler—note that several compilation passes, including array concatenation and array reuse, are not type-preserving.

## 10 RELATED WORK

*Enforcement of speculative constant-time.* There is a large body of work that develops automated techniques for detecting speculative leaks. These works use a variety of approaches, including type systems, symbolic execution, or fuzzing, and have a broad range of targets, including source programs, intermediate representations, assembly, or binary code. We refer the reader to the survey [25] for further information and an overview of the tools up to 2022.

From the point of view of this paper, the most important distinction among these works is the security definition they target. In this work, we consider speculative constant-time, and require that programs do not leak through observations. An alternative is to require that speculative execution does not leak more than sequential execution. We refer to this policy as relative speculative constant-time (RSCT), although the terms relative non-interference [25] or speculative non-interference [34] are sometimes used in the literature.

RSCT can be defined as an instance of SCT, by letting the relation $\phi$ be the maximal relation between memories for which sequential execution of a fixed program $c$ coincide. It follows that a compiler pass preserves RSCT iff it preserves SCT for an arbitrary relation $\phi$ and reflects CT for an arbitrary relation $\phi$. However, it is unclear that all transformations of interest reflect constant-time. In general, RSCT and all conditional policies (i.e., policies stated as implications) seem less amenable to preservation proofs.

*Preservation of constant-time.* Over the last few years, researchers have developed new techniques to prove that compilers preserve constant-time. The first proof of preservation of constant-time introduces CT-simulation, a technique that extends the usual commuting diagram of forward simulation from compiler correctness proofs into a commuting cube, that involves two related executions at source level and two corresponding executions at target level [17]. The technique, while general, is overly powerful for many common optimizations of interest. Therefore, subsequent works explore a simpler technique, which consists in proving that the leakage of a target program can be computed from the leakage of the corresponding source program—and potentially the public part of the initial state. This suffices to establish, using a simple argument, that the compiler preserves constant-time. This technique (in some variant forms) was used to verify formally that the CompCert and the Jasmin compilers, which constitute two prime examples of formally verified compilers, preserve constant-time [15, 18].

*Preservation of speculative constant-time.* Patrignani and Guarnieri [49] were among the first to explore systematically the interactions between compilers and Spectre attacks. Similar to ours, their exploration is contrasted. On the negative side, they observe that existing compilers, including

compilers that implement SLH, generate code that leaks speculatively. On the positive side, they show that with adequate care it is possible to generate code that is speculative constant-time. However, their work does not consider common compiler optimizations as done in this paper.

An alternative to preservation of speculative constant-time is to implement compiler-based mitigations that make programs speculative constant-time. Blade [57] is a prime example of compiler-based mitigation that has been used to protect cryptographic code written in WASM against Spectre v1. Informally, Blade identifies secrets as sources and leaking instructions as leaks, and implements an efficient algorithm for the Max-Flow/Min-Cut problem to achieve optimal placement of protections.

*Secure compilation.* Secure compilation is a research area that focuses on the interactions between compilation and security. Since compilers are complex objects and security policies are more intricate than trace-based properties, many of the results in this area are formalized within a proof assistant. One main line of work aims to develop a framework to define, analyze and contrast rigorously different notions of secure compilation [1, 2, 48]. The focus of these works is definitional, and therefore their general aim is generality rather than on specific compilers or properties. Another main line of work focuses on specific compilers and specific properties, including various forms of safety and integrity [20, 28, 44, 45, 67], and resource usage [7, 21, 23, 33, 47, 61]. Other works approach secure compilation from a broader perspective, and in particular considers compiler security in a broader context [29, 55, 64].

## 11 CONCLUSION

Our paper introduces a framework for proving preservation of speculative constant-time, and shows that the framework can be instantiated to many common optimizations from the literature. Our overall conclusion is that preservation of speculative constant-time for a realistic compiler like Jasmin is a plausible target, provided sufficient care is taken to align the source-level notion of security with compiler passes that may introduce additional leakage. A natural direction for future work is to carry a more systematic exploration of source-level type systems, in order to achieve the best performance of generated code. Another direction for future work is to extend our results to probabilistic programs, which satisfy a probabilistic notion of speculative constant-time, and to programs which use declassification.

## ACKNOWLEDGMENTS

This research was supported by the *Deutsche Forschungsgemeinschaft* (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; and by the *Agence Nationale de la Recherche* (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006.

## REFERENCES

[1] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 14:1–14:48. https://doi.org/10.1145/3460860

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 256–271. https://doi.org/10.1109/CSF.2019.00025

[3] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*

*(Lecture Notes in Computer Science, Vol. 9665)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, 622–643. https://doi.org/10.1007/978-3-662-49890-3_24

[4] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 526–540. https://doi.org/10.1109/SP.2013.42

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 965–982. https://doi.org/10.1109/SP40000.2020.00028

[7] Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2013. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis—Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8552)*, Ugo Dal Lago and Ricardo Peña (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

[8] Nadav Amit, Fred Jacobs, and Michael Wei. 2019. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 285–300. https://www.usenix.org/conference/atc19/presentation/amit

[9] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1753–1770. https://doi.org/10.1109/SP46215.2023.10179355

[10] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1094–1111. https://doi.org/10.1109/SP46215.2023.10179418

[11] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-grained Constant-time Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 83–96. https://doi.org/10.1145/3548606.3560689

[12] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 225–242. https://doi.org/10.1145/3372297.3417268

[13] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 777–795. https://doi.org/10.1109/SP40001.2021.00008

[14] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1267–1279. https://doi.org/10.1145/2660267.2660283

[15] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[16] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1884–1901. https://doi.org/10.1109/SP40001.2021.00046

[17] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 328–343. https://doi.org/10.1109/CSF.2018.00031

[18] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 462–476. https://doi.org/10.1145/3460120.3484761

[19] Daniel J Bernstein. 2005. Cache-timing attacks on AES.

[20] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. 2019. Compiling Sandboxes: Formally Verified Software Fault Isolation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 499–524. https://doi.org/10.1007/978-3-030-17184-1_18

[21] Sandrine Blazy, André Oliveira Maroneze, and David Pichardie. 2013. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8164)*, Ernie Cohen and Andrey Rybalchenko (Eds.). Springer, 281–303. https://doi.org/10.1007/978-3-642-54108-7_15

[22] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[23] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom—June 09—11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 270–281. https://doi.org/10.1145/2594291.2594301

[24] Chandler Carruth. 2018. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.

[25] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 666–680. https://doi.org/10.1109/SP46214.2022.9833707

[26] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 913–926. https://doi.org/10.1145/3385412.3385970

[27] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security*.

[28] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 292–307. https://doi.org/10.1109/SP.2014.26

[29] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 73–87. https://doi.org/10.1109/SPW.2015.33

[30] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27. https://doi.org/10.1007/S13389-016-0141-6

[31] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 1690–1704. https://doi.org/10.1145/3576915.3623112

[32] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 845–858. https://doi.org/10.1145/3133956.3134029

[33] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 204:1–204:29. https://doi.org/10.1145/3428272

[34] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1–19. https://doi.org/10.1109/SP40000.2020.00011

[35] Lorenz Hetterich, Markus Bauer, Michael Schwarz, and Christian Rossow. 2024. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*, Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro Cardenas (Eds.). ACM. https://doi.org/10.1145/3634737.3637662

[36] Intel Labs. 2018. Retpoline: a software construct for preventing branch-target-injection. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf.

[37] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 632–649. https://doi.org/10.1109/SP46214.2022.9833713

[38] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5747)*, Christophe Clavier and Kris Gaj (Eds.). Springer, 1–17. https://doi.org/10.1007/978-3-642-04138-9_1

[39] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2038–2052. https://doi.org/10.1145/3576915.3616611

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002

[41] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1109)*, Neal Koblitz (Ed.). Springer, 104–113. https://doi.org/10.1007/3-540-68697-5_9

[42] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, Christian Rossow and Yves Younan (Eds.). USENIX Association. https://www.usenix.org/conference/woot18/presentation/koruyeh

[43] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 42–54. https://doi.org/10.1145/1111037.1111042

[44] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 395–404. https://doi.org/10.1145/2254064.2254111

[45] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[46] Santiago Arranz Olmos, Gilles Barthe, Chitchanok Chuengsatiansup, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. 2024. Protecting cryptographic code against Spectre-RSB. Cryptology ePrint Archive, Paper 2024/1070. https://eprint.iacr.org/2024/1070 https://eprint.iacr.org/2024/1070.

[47] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29. https://doi.org/10.1145/3341687

[48] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 392–404. https://doi.org/10.1109/CSF.2017.13

[49] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 445–461. https://doi.org/10.1145/3460120.3484534

[50] Colin Percival. 2005. Cache missing for fun and profit.

[51] Antoon Purnal. 2024. PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity. https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber-ml-kem-to-improve-pqc-implementation-maturity/

[52] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. 2018. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1397–1414. https://doi.org/10.1145/3243734.3243775

[53] Michael Scott. 2024. Compiler-introduced timing leaks—FIAT-Crypto. Posted on PQC-Forum, June 23, 2024.

[54] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. 2024. TeeJam: Sub-Cache-Line Leakages Strike Back. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024, 1 (2024), 457–500. https://doi.org/10.46586/TCHES.V2024.I1.457-500

[55] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 1–15. https://doi.org/10.1109/EuroSP.2018.00009

[56] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (2010), 37–71. https://doi.org/10.1007/S00145-009-9049-Y

[57] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330

[58] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1491–1505. https://doi.org/10.1109/SP46214.2022.9833570

[59] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 679–697. https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen

[60] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. 2023. DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2306–2320. https://doi.org/10.1109/SP46215.2023.10179326

[61] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL (2019), 62:1–62:30. https://doi.org/10.1145/3290375

[62] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 3825–3842. https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[63] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[64] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3655–3672. https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao

[65] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.* 7, 2 (2017), 99–112. https://doi.org/10.1007/S13389-017-0152-Y

[66] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 7125–7142. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh

[67] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: fully verified software fault isolation. In *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister (Eds.). ACM, 289–298. https://doi.org/10.1145/2038642.2038687